Speculative execution ignoring privilege-level changing instructions on ARM64

Verified on CPUs
- Cavium ThunderX2 T99
- Cortex A-72 in Raspberry Pi 4

Mechanism
ARMv8 exception-generating instructions (BRK, HLT, HVC, SMC and SVC) and the exception-return instruction (ERET) behave unexpectedly at least on some ARM CPUs. Even though all these instructions change the CPU privilege-level (unless they lead to a fault), and they always cause a jump to another address, ARM CPUs speculatively execute instructions that follow them as if the privilege-level changing instructions were not jump instructions.

The speculative execution does not cross privilege-levels (to the jump target as one would expect), but it continues on the current privilege level as if those instructions did not change the control flow. Later, the results of this speculative execution are always architecturally discarded.

This speculative execution is very reliable (seems to be unconditional) and it manages to complete even relatively performance-heavy operations. For example we flushed a function code from the cache using "dc civac" and "dsb sy", but the speculative execution still managed to fetch that function from the main memory and execute it even though it was behind an SVC instruction (leading to an infinite FUTEX wait) and therefore architecturally unreachable. It is naturally possible to observe the microarchitectural side effects of that speculative execution using e.g. the FLUSH+RELOAD technique.

Scenarios tested

a) SVC instruction
https://github.com/google/safeside/blob/master/demos/speculation_over_syscall.cc
We tested the SVC instruction directly in the userspace code using the following three steps:

1) FLUSH operation (FLUSH+RELOAD technique) on a memory oracle (using "dc civac" in a loop and "dsb sy" in the end) that is used for cache-hit measurements.
2) Main thread creates a new thread that forever blocks on an infinite FUTEX ("sys_futex" syscall invoked by the SVC instruction) that is followed by an architecturally unreachable access to that oracle.

Code of the function executed by the new thread:

```c
int *val = new int;
val[0] = 23;
asm volatile(
    "mov x0, %0
    "mov x1, %1
    "mov x2, %2
    "mov x3, %3
    "mov x4, %4
    "mov x5, %5
    "mov x8, %6
    "svc #0":
    "r"(val),"r"(FUTEX_WAIT),"r"(22),"r"(nullptr),
    "r"(nullptr),"r"(0),"r"(__NR_futex));
ForceRead(isolated_oracle.data() +
static_cast<size_t>(data[unsafe_offset]));
```

3) The main thread waits a second and then performs the RELOAD operation (FLUSH+RELOAD technique) on the oracle and verifies that the data[unsafe_offset] index was speculatively accessed by the blocked thread, because it was the only cached index in the oracle.

b) ERET, HVC and SMC instructions

https://github.com/google/safeside/blob/master/demos/eret_hvc_smc_wrapper.cc
https://github.com/google/safeside/blob/master/kernel_modules/kmod_eret_hvc_smc/eret_hvc_smc_module.c

We tested the EL1 instructions (ERET, HVC and SMC) in a kernel module on a speculative path that was reached from the user space using a syscall.

In the kernel code we placed the instruction into an unreachable path and steered the speculative execution to it using the "likely" hint. We put there unconditionally the instruction (ERET, HVC or SMC) followed by a memory access to an address in the memory oracle provided in the syscall argument.

The syscall handler code:

```c
asmlinkage int sys_fetch_after_eret(const char __user *address) {
    size_t * memory;
```
Using the same FLUSH+RELOAD technique we can again verify that the address provided by the userspace was speculatively accessed by the kernel.

c) BRK and HLT instructions

https://github.com/google/safeside/blob/master/demos/speculation_over_sw_breakpoint.cc

We tested the breakpoint instructions (BRK and HLT) analogically to ERET, HVC and SMC in a speculative branch, we just executed the same gadget in a normal user space program. We also retested the SVC instruction this way.

d) Meltdown-UD

https://github.com/google/safeside/blob/master/demos/meltdown_ud.cc

Finally we were able to generalize the speculative behavior of ARM CPUs using an invalid instruction opcode that leads to an UndefinedInstructionException. Even that instruction did not stop the speculative execution which is called Meltdown-UD in this paper: https://arxiv.org/pdf/1811.05441.pdf. We verified this vulnerability both speculatively (using the same technique as when testing ERET, HVC and SMC instructions) and non-speculatively using a SIGILL signal handler which moved the program counter after the speculative read.

Main function:

```c
...  
asm volatile(".word 0xf7f0a000"); // Invalid instruction, raises SIGILL.  
// Architecturally unreachable code.  
ForceRead(isolated_oracle.data() +  
static_cast<size_t>(data[unsafe_offset]));  
std::cout << "Dead code. Must not be printed." << std::endl;  
// SIGILL signal handler moves the instruction pointer to this label.  
asm volatile("afterspeculation:");  
...  
```
SIGILL signal handler:

```c
static void sigill(
    int /* signum */, siginfo_t * /* siginfo */, void *context) {
    ucontext_t *ucontext = static_cast<ucontext_t *>(context);
    ucontext->uc_mcontext.pc = reinterpret_cast<greg_t>(local_handler);
}
```

Local handler:

```c
static void local_handler() {
    asm volatile("b afterspeculation");
}
```

**Security implications**

This behavior of ARM CPUs basically creates a new Spectre gadget class. It leads to accidental speculative concatenations of code sections that cannot be executed one after the other architecturally. Those concatenations can create new Spectre v1 or Spectre v4 gadgets. E.g., any code that is accidentally linked after an ERET instruction in the kernel will be speculatively executed anytime the control flow approaches that ERET instruction - unless that following code contains (again by accident) a speculative-execution-blocking instruction (such as the "dsb sy") that would stop that. If it contains a branch instruction, the speculative execution will be naturally steered to its branch target. The same holds for all the other affected instructions.

**Software mitigations**

Proper software mitigation of this vulnerability is to insert a speculative-execution-blocking instruction after any privilege-level changing instruction. This might be relatively straightforward and without performance implications in the kernel code, but e.g. mitigating all occurrences of the SVC instruction would require a userland library (e.g. GLIBC) update, because in that case the kernel code is functionally irrelevant.